MUSBUS: Experiences Using a Terminal Emulator to Select NAS Computer Systems

E. N. Miya
T. Woodrow
NAS Systems Division
NASA Ames Research Center
Moffett Field, CA 94035

Report RND-91-010, July 1991

eugene@nas.nasa.gov
woodrow@nas.nasa.gov
The last diagram needs to have a line hand-drawn in the margin

ABSTRACT

MUSBUS [Monash University Suite for Benchmarking UNIX Systems] is a terminal emulation program to time and "stress" computer system performance. MUSBUS simulates multiple human users on an interactive computing environment executing a workload script.

MUSBUS offers benchmark control and stress capabilities lacking in an environment like a bare operating system. One program simulates user keyboard activity, and another distributes resource utilization (random or not) during a benchmarking session.

This report documents the qualitative experience and some results of using MUSBUS. This report discusses the needs, problems, some solutions, and the advantages to this performance measurement approach. This report is not a guide to using MUSBUS since that function is better served by MUSBUS documentation [3].

Introduction

The Numerical Aerodynamic Simulation (NAS) Program is the National supercomputer resource for the aerospace industry. NAS is located at the NASA Ames Research Center on the San Francisco Bay in the heart of Silicon Valley. Like many labs, NASA Ames has hundreds of different computer systems. Like many facilities, NAS buys state-of-the-art support processors and gets the best machines for the dollar measured against a representative workload.

To accomplish this, NAS develops benchmarks and workloads to select computers. One representative requirement is processor performance which supports interactive terminal sessions. Therefore, NAS needs an interactive terminal session benchmark.

MUSBUS

MUSBUS [Monash University Suite for Benchmarking UNIX Systems] is a set of programs developed by Ken McDonell [2,3] to control workload-based performance measurement. The programs emulate a user typing commands with resulting terminal output. MUSBUS allows a benchmarker to control relevant variables like the number of users while handling extraneous details. MUSBUS is not a single program, and it gives no single-figure-of-merit.

MUSBUS is a better benchmarking environment than a bare-bones operating system. MUSBUS allows a benchmarker to concentrate on issues relevant to performance measurement rather than be bogged down in shell scripts, resource contention, and other operating problems. MUSBUS is not perfect; it adds an invasive overhead, and requires additional time to port, but it is very useful.

MUSBUS, Our Workload, and Everything

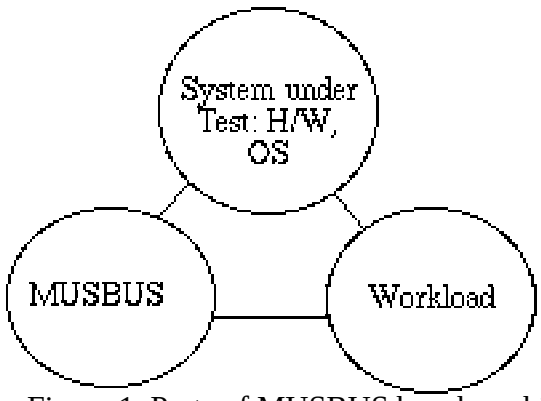


Figure 1. Parts of MUSBUS benchmarking.

This simple diagram illustrates three parts of MUSBUS benchmarking. These parts show where testing problems can appear. New systems under test always have bugs. The authors found small problems with MUSBUS software, and our workload, called `Work-`

`load.nas`, has syntactic and semantic portability problems. All three parts must work lest MUSBUS dies or takes bad data. Despite these problems, NAS is a MUSBUS convert; emulation has elegance.

Major MUSBUS parts

MUSBUS has three components (excluding workload):

- shell variables
- files: programs and data
- directories

Shell variables control workload execution and are found in Makefiles. They allow simple control of how many users are simulated (e.g., NUSER variable). OSTYPE helps control portability. Experienced MUSBUS users will control other variables. ITER, RATE, TTYS, DIRS, SCRIPTS, SHELL, and so forth. The experience using these variables is deferred to the section on **Porting and Using MUSBUS**.

Files: Three files require attention:
1) Makefiles -- Makefiles control MUSBUS, its Workloads, and system portability. They compile and run the system. MUSBUS's Makefile contains OSTYPE and other variables which are passed to *run*. All Workloads require a Makefile as well.

2) Run -- Run is the shell script which executes the Workload. It actually controls the measurement, times it, and cleans up.

3) *.awk -- Awk reformats output report measurements like timing. Awk keeps MUSBUS portable and flexible. The choices are set by OSTYPE shell variable.

Additionally, a fourth program may interest the user:

4) Keyb -- Keyboard simulation -- Keyb reads an input script and meters the script out at a specific, adjustable speed. Keyb is a simple simulation of a human typist. Keyb implicitly controls the shell command execution rate and the explicit input rate for interactive commands like text editors.

New MUSBUS users can skip the details of advanced commands executed in the `run` script. Advanced commands include:

Mkscript -- make workload scripts -- Mkscript takes a `script.master` workload and permutes duplicate scripts [named `script.1, script.2, ...`]. Job steps in these scripts are executed in random order. This distributes processes using locks and files further reducing contention.

Makework -- generate work -- Makework controls the execution of a simulated user running a workload script. It sets up the execution environment and runs the keyb program's implict shell control.

Most MUSBUS users do not require changing these commands, but we did debug them.

Directory structure is simple:
Workload contains executables, data, and control information for a single simulated user running a Workload. Multiple copies are made and run. The workload is specified by the script.master and Makefile. A default Monash Workload provides testing and debugging. Two additional workloads come with MUSBUS to simulate "text processing" and "stress."

Tmp contains subdirectories to simulate users. This isolates the workspace of simulated users. The permutations of the workload can spread the work.

Result contains timing and command logs. A quick view of the MUSBUS shows:

```
BSD4v2time.awk  BSDtime.awk     Groan*          Intro.lp
Intro.nr        M.apollo        M.att           M.dec_risc
M.hp            M.ibmrisc       M.mips          M.mot
M.sun           M.vax           MAKE*           Makefile
README.SPEC     README.orig     SysVtime.awk    TEMPLATE
Tmp/            Workload/       cctest.c        check.sed
cleanup*        clock*          clock.c         clock.c.sv
fixPATH*        fs.awk          getwork.c       iamalive*
iamalive.c      keyb*           keyb.c          limit*
limit.c         makework*       makework.c      makework.h
mem.awk         mkperm*         mkperm.c        mkscript*
mkscript.out*   musbus.1        result/         run.spec*
signature*      time.awk        ttychk*         ttychk.c
util.c          work.c*
```

Specifically note: the executables are noted using '*' and the directories are noted '/'. A sample Workload contains:

```
DESCRIPTION     cleanstderr*    edscr2.dat      script.3
Makefile        dummy.c         grep.dat        script.4
cat.dat         edit.dat        script.1        script.master*
cctest.c        edscr1.dat      script.2        script.out
```

What MUSBUS does

The MUSBUS program does six basic functions:
1) Checks single-copy, workload execution
2) Meters output volume and rate
3) Prepares simulated user directories
4) Waits one minute
5) Runs a test based on desired test conditions
6) Cleans up

MUSBUS first runs a single copy of the master script to syntax check and size output. If the workload has an error, MUSBUS stops, drops error messages into "script.out.bad" and "results/log."

If the single copy workload is successful, MUSBUS creates individual test user directories in Tmp. Prior to full workload execution, MUSBUS waits sixty seconds to stabilize system

activity.

MUSBUS then forks simulated user processes (controlled by the keyb command). Each user executes a prologue before executing a randomized workload (randomized script.master). The prologue contains commands requiring some priority (like global variable initialization).

Lastly, the MUSBUS workload epilog cleans and removes the temporary directories created before the test. Timing and error messages are stored in result/log files. MUSBUS concatenates runs, KENBUS clobbers old runs.

The NAS SPS Workload

NAS must provide front-end support to supercomputer users. This requires intensive I/O and character processing performance for editing, network communcation, and some file storage. The support is not floating-point intensive, but users still need good price performance. A support processor must compete favorably with a modern single-user workstation.

A support processor system must serve 1600 users, and a given computer may handle 100-200 users peak. The number of processors and their speed can vary, but each user requires between 10-20 MBs of disk (16..32 Gbytes total). A workload to simulate 128 users (a lot) was selected.

`Workload.nas` was built after a one week observation period of daily command logs on existing support processors. Real workload data was collected using BSD *sa*(8) and Amdahl UTS acctcom(8) in preparation for a representative workload. `Workload.nas` was proportioned to match the real workload.

The following table shows the ten most frequently executed commands per processor by relative frequency. The last entries are the ten most frequently executed commands in the MUSBUS `Workload.nas`. Note that MUSBUS most closely matches the load on the host `prandtl`.

amelia		fred		orville	
% freq	proc	% freq	proc	% freq	proc
14.3	test	31.6	test	14.0	csh
11.3	echo	8.1	csh	9.1	sendmail
5.6	csh	7.3	echo	7.9	cron
3.8	grep	5.7	awk	7.4	sh
3.4	sh	5.0	grep	4.9	awk
3.4	gethome	3.3	sed	3.9	echo
3.2	rm	3.0	expr	3.4	test
3.1	cron	2.7	cron	2.9	grep
3.0	awk	2.2	sh	2.6	lpd
2.9	rcp	2.1	hostname	2.6	expr
54.0		71.0		58.7	

wilbur	prandtl	MUSBUS

% freq	proc	% freq	proc	freq	cmd
12.8	csh	7.4	sendmail	10.0	mail
6.8	cron	7.1	sh	10.0	csh
6.7	sh	5.6	sed	9.0	sh
6.1	echo	4.6	cat	8.0	rcp
5.4	sendmail	4.6	rm	8.0	pwd
4.2	grep	4.3	basename	6.0	date
4.0	expr	4.3	date	6.0	sed
3.0	date	3.8	awk	5.0	awk
2.9	test	3.6	rcp	5.0	basename
2.7	rm	3.0	pg	5.0	rm
54.6		48.6		72.0	

Some workload commands require data, so reasonable simplifications are required generated, for example:
man(1) asks for the *sh*(1) man page. Seemed reasonable; until a system without an *sh* man page was found.
Mail's input is a 20-line file (our simulated user sending his latest discovery to a colleague).
The workload has one edit (*ed* for a variety of reasons since this test is keystroke-bound).
The workload has token compilations.
Size is a distinguishing feature of program text for editing, compilation, and even copying. A typical CFD program (ARC2D, part of the PERFECT Club Suite) contains 16,000 lines of code (0.5 MB). Simulating 128 users editing a code this large is non-trivial. Systems under test required lots of disk.

The command mix is placed into "script.master" in `Workload.nas` which is also filled with data and non-system commands. The complete `Workload.nas/script.master` is listed in Appendix A. MUSBUS tries to execute and count commands, but it miscounts

1) shell variable initializations
2) pipe commands
3) indirectly executed commands (in shell scripts or forked e.g.,*sendmail*).

This accounts for the difference between the final process count between the `script.master` and `result/log`:

```
Workload script profile: shell=/bin/sh
62 commands (header:3 & trailer:2)
freq  command    freq  command    freq  command
9     mail.csh   6     rcp        6     rm
5     cat        5     sed        4     awk
4     date       3     basename   2     ls
2     ping       2     pwd        1     chmod
```

```
1     comp.sh    1     echo       1     ex
1     export     1     find       1     grep
1     man        1     mkdir      1     printenv
1     rsh        1     set        1     sort
1     touch
```

Porting and Using MUSBUS

MUSBUS ports like any other program: First copy it. Edit as needed. Compile (type `make`). Test it. If the program dies or produces bad results, repeat the last three steps. Adapt for local use (edit workload, debug, and test). Using MUSBUS was actually fun, because it was:

1) easy to use
2) easy to modify (well-structured and modular)
3) easy to debug (usually fairly simple)

MUSBUS's design makes it impressive.

MUSBUS's modularity is perhaps the most impressive part of the system. Superb portability is attained by *awk* programs, shell variables and scripts. The C, shell, and *awk* programs have the right functionality to minimize run time overhead yet give flexibility and power. A benchmarker can isolate and modify these scripts as needed.

A bare bones operating system requires benchmarkers to write their own scripts. Subtle UNIX differences on every machine force debugging customization. This burns development time. MUSBUS covers these subtle portability and reliability problems. Viewing the issues in detail:

File copying: Our first problem is simply moving MUSBUS and its workloads to the system under test. Electronic mail, network file transfer, or tape are used. In our experience, one vendor had a byte ordering problem which took additional time to filter. Nothing is quite as simple as it seems.

Editing: The only significant pre-compile editing is setting a Makefile shell variable OSTYPE. Setting OSTYPE to SysV or 4BSD simplifies portability using *awk* scripts. Vanilla UNIX systems are handled this way. Exceptions are fairly easily edited, for instance, the *time*(1) command on a Cray Y-MP includes extra fields detailing the "clock period" count. The file time.*awk* is easily edited to handle this.

Compilation: Just type *make*. This usually works. Default parameters should execute one iteration of a one user default Workload.

Test run: A single command called *run* handles details or *make* can call *run*.. MUSBUS comes with three test workload directories: `Workload`, `Workload.text`, and `Workload.stress`. Moving (renaming) directories or setting a `workdir` variable in the Makefile selects the test environment. If a benchmarker lives a clean life, MUSBUS is ready.

Debug problem: Subtle system-under-test variations might prevent first time execution.

The *run* shell script has a subtle inconsistency across different systems. A problem iterating runs counts *n* or *n-1* iterations. Some systems run three iterations when others would run two. A semantic inconsistency exists across systems. This bug was not tracked for lack of time, so quick hack ran one iteration. Similarly, value lists, or ramps, did not work on some systems, so each data value was run "by hand."

C program initialization was a problem for one structure field in getwork(). Only one machine found the lack of explicit initialization to 0, but it could prove serious. Another problem was that *awk* scripts did not read hour-long benchmarks. Fortunately, most MUSBUS problems were attributable to workloads or shell variables.

McDonell advises debugging with `nusers`=1 and `iterations`=1. This is useful, and it also helps to create a nearly empty workload for debugging, perhaps containing one simple command like *echo*. An empty workload debugs and executes faster than the default Workload.

Repeat compilation and test until MUSBUS works.

Adapt for local use: Use `Workload.nas`. The problems are syntactic and semantic. Once the workload is running, the focus is shell variable control. MUSBUS attempts to run a single copy of a workload prior to taking a measurement. If a workload fails, it places information into "script.out.bad" in the workload directory. (Nice touch.)

Syntax problems: workload syntax problems require exposure to a real system.

Commands like

```
ping 1 1 $host
```

or

```
ping -c1 -s1 $host
```

behave inconsistently across systems.

Semantic problems: permission and security present a clash of problems. One vendor had a strange `umask` setting (i.e., 153).

Benchmarkers do not encounter all problems on small-scale runs. Larger runs require root permission to exceed the per user process limit, but workloads containing network commands, e.g., *rcp*, fall into a testing twilight zone. Root isn't just any user, and many systems prevent root from *rcp*ing.

Simulating many users tends to fill file systems (i.e., mailboxes overflow); simulated users doing simulated editing require really big files to edit. `/tmp` requires adequate storage. This is a system-under-test problem, but it can cause MUSBUS failure.

Extensive I/O is a problem. MUSBUS is designed to handle this beautifully, but we redirected test output to /dev/null during measurement for between-system consistency. Most SPS work is I/O bound. Thus time is the only measured quantity (real, CPU, system and user).

Once these problems are solved, the really interesting measurement problem begins. The user load (NUSERS) is the first interesting variable, and the benchmarker can vary this to his or her heart's content. We use other MUSBUS control variables including: NSCRIPTS, ITERS, DIRS, TTYS, and RATE.

NSCRIPTS improves workload permutation. This helps resource sharing by spreading the users across different workload steps (processes).

ITERS was set to '1' after learning workload duration. Individual timing did not vary significantly in our tests.

DIRS spreads files across multiple disks and thus balances some of the I/O load. This is useful for tests requiring storage.

TTYS is the most interesting variable. Directing output to /dev/null during measurement achieves greater consistency between systems. During development, however, output was directed to as many as eight real or pseudo-terminals. TTYS is useful and interesting terminal benchmarks can use this variable.

RATE (characters per second of simulated typing) is useful for debugging. RATE=8 allows faster command execution over the default 3 cps.

Tuning MUSBUS for the System-Under-Test

Workload.nas is an extremely difficult workload for simulating 128 users. Workload.stress is light compared to Workload.nas. Workload.nas does not run on UNIX systems lacking sufficient hardware and tuning. System performance tuning falls into two categories: machine-independent and machine-dependent tuning.

Machine-independent tuning is possible with a common operating system, but default environmental assumptions vary widely. Workstation vendors do not tune for 128 users on their machines. Every vendor increased process table size (the NPROC parameter). A similar adjustment is the open-file table size (NFILE). This information is easily shared between vendors, and these are examples.

Machine-dependent tuning usually implies a multiprocessor architecture. Multiprocessors have unique architectures. Tuning was not easily shared between vendors. For instance, the SGI 4D-380 "Predator" had constants governing "spin-locks" which are not likely to be found on other processors. Vendors suggested adjustments, and the benchmarkers noted changes.

Finally, hardware configuration also contributes to performance. Vastly differing I/O systems produced striped and non-striped file systems. At best, our decisions attempt to favor the highest possible performance.

Interpreting the Results

Vendor specific results appear in Appendix B. McDonell provides some guidance for interpreting results. He suggests taking elapsed time and dividing in the user load. The NAS SPS workload must run a heavy user load. The selected NUSER input ramp was

1 2 4 8 12 16 24 32 40 48 56 64 80 96 112 128 users.

From this domain, a range of MUSBUS measurements was made.

Elapsed time for a complete workload run is the first MUSBUS measure. A benchmark can determine average command execution time or execution time per user. These plot the number of commands executed per second against the number of simulated users on a DEC VAX 6000 multiprocessor. Appendix section B.2 shows results for an SGI 4D/380 with an *awk* script bug.

Conclusions

A separate, detailed market survey [4] is available describing tested systems. This short report concentrates on observations and experiences using MUSBUS to simulate user activity (terminal emulation) in a time-shared environment. The report draws three conclusions:

The value of terminal emulation

Terminal emulation is a useful method for controlling benchmarks and workloads. It is useful for interactive, time-sharing environments like workstations and medium-sized mainframes where users simultaneously execute many commands.

The value of MUSBUS

MUSBUS offers better control compared to a bare operating system or other terminal emulators. It has simple resource management, workload and shell variable control. A benchmarker concentrates on measurement detail rather than a complex control program.

The value of the NAS SPS Workload

The NAS SPS Workload adequately describes our need for a support processor in 1991, but the workload has limited value after our procurement. Our workload was developed after a mere week of coarse sampling. Representativeness deserves some place over expedience. This workload lacks floating-point, uses few compilations, and consists of UNIX utilities. This is what our support processors do. The workload ran without terminal I/O for consistency. Our actual workload will without a doubt change.

The one observed distressing problem is social rather than technical. Many benchmarkers are dependent on the standard default Workload. This Workload was developed for McDonell's Monash University. More Workloads are needed to provide more experience. Their development requires long-term, high-quality monitoring. Other sites should develop their own workloads rather than use NAS SPS or Monash's. Ken McDonell notes that Monash University fully instrumented their bin directories to capture all parameters. These kinds of studies are needed.

One last piece of advice to benchmarkers: make certain that results measured remotely are duplicated on delivery.

References

[1] David Hinnant, "Performance Measures," *UNIX Review*, vol. 8, no. 12 (December 1990), pages 34-40.

[2] Ken J. McDonell, "Taking Performance Evaluation out of the "Stone" Age," *Conference Proceedings Summer 1987 Usenix Meeting*, Phoenix, AZ, 1987.

[3] Ken J. McDonell, "An Introduction to the Monash Benchmark Suite (MUSBUS)," Technical Report, Monash University, Clayton, Aust., May 1988.

[4] Thomas Woodrow, "Support Processing Subsystem/Scientific Analysis Subsystem Market Survey," TR RND-91-002, January 1991.

[5] FIPS. *Guidelines for Benchmarking ADP Systems in the Competitive Procurement Environment*, FIPS PUB 42-1, US. Department of Commerce, National Bureau of Standards, May 1977.

Appendix A. `Workload.nas/script.master`

```
%W% /bin/sh -ie
PATH=XXX:$PATH:/usr/local/lang:/usr/etc:.
suf=`pwd`
suff=`basename $suf`
remote=radon
mailtarget=eugene
export PATH remote mailtarget suff
mkdir /tmp/$$ tmp
touch tempfile
%
%% 1 edit
%
date
mail.csh $mailtarget
keyb edscr1.dat | ed arc2d.f
%
%% 2 chmod, rm
%
chmod u+w tempfile
rm tempfile
basename `pwd`
mail.csh $mailtarget
%
%% 3 man, rm
%
man sh > /tmp/shell.man$suff 2>/dev/null
rm /tmp/shell.man$suff
mail.csh $mailtarget
%
%% 4 nroff, ping
%
% nroff -man /usr/man/man1/sh.1 > /dev/null
ping -c1 -s1 $remote
mail.csh $mailtarget
%
%% 5 ping, rcp stuff
%
ping -c1 -s1 $remote
 rcp mflops90.f $remote\:/tmp/dummy1.$suff
rcp $remote\:/tmp/dummy1.$suff dummy1
 rcp mflops90.f $remote\:/tmp/dummy2.$suff
rcp $remote\:/tmp/dummy2.$suff dummy2
 rcp linpackd.f $remote\:/tmp/dummy3.$suff
rcp $remote\:/tmp/dummy3.$suff dummy3
ls -CF
% diff ./arc2d.f ./dummy.arc2d >/dev/null
```

```
rsh $remote -n rm /tmp/dummy1.$suff /tmp/dummy2.$suff /tmp/
dummy3.$suff < /dev/null
rm dummy1 dummy2 dummy3
mail.csh $mailtarget
%
%% 6 date, awk, sed
%
date
awk '{print $0}' mflops90.f   > /dev/null
sed -e '/^[Cc]/d' mflops90.f   > /dev/null
mail.csh $mailtarget
%
%% 7 cat + rm
%
cat mflops90.f > ARC2D.F
rm ARC2D.F
mail.csh $mailtarget
%
%% 8 awk, cat, tr, grep, wc, sed
%
awk '{print $0}' mflops90.f > /dev/null
cat mflops90.f | tr a-c A-Z | grep '^C' | wc -l
sed -e '/^[Cc]/d' mflops90.f > /dev/null
mail.csh $mailtarget
%
%% 9 compiles
%
which f77
which cc
comp.sh
%
%% 10 basename, pwd, date, cat and rm
%
basename `pwd`
date > date.out
cat date.out
rm date.out
%
%% 11 pwd, ls, diff
%
pwd
ls -CF
% diff ./linpacks.f ./linpackd.f > /dev/null
%
%% 12 sort, find
%
sort sortfile > /dev/null
find /tmp/lvl -name passwd -print
%
```

```
%% 13 grep, sed
%
sed -e '/^[Cc]/d' mflops90.f > /dev/null
%
%% 14 awk, sed, ls
awk '{print $0}' mflops90.f > /dev/null
sed -e '/^[Cc]/d' mflops90.f > /dev/null
%
%% 15 awk, sed
%
awk '{print $0}' mflops90.f > /dev/null
sed -e '/^[Cc]/d' mflops90.f > /dev/null
%
%% 16 cat, grep
%
cat mflops90.f | tr a-c A-Z | grep '^C' | wc -l
grep '^C' mflops90.f > /dev/null
%
%% 17 status
%
set
printenv
pwd
basename `pwd`
date > date.out
%
%% 18 mail
%
mail.csh $mailtarget
cat mail.csh >/dev/null
%
%%
rm -rf tmp /tmp/$$
echo "***** All Done *****"
```

Appendix B. Vendors Tested

Vendors tested for the SPS were noted for
1) communication ease (data transfer)
2) ease to reboot
3) ease of reconfiguration
4) other problems/solutions/observations
See the market survey by Woodrow for more specific details[4]. The order presented here is chronological.

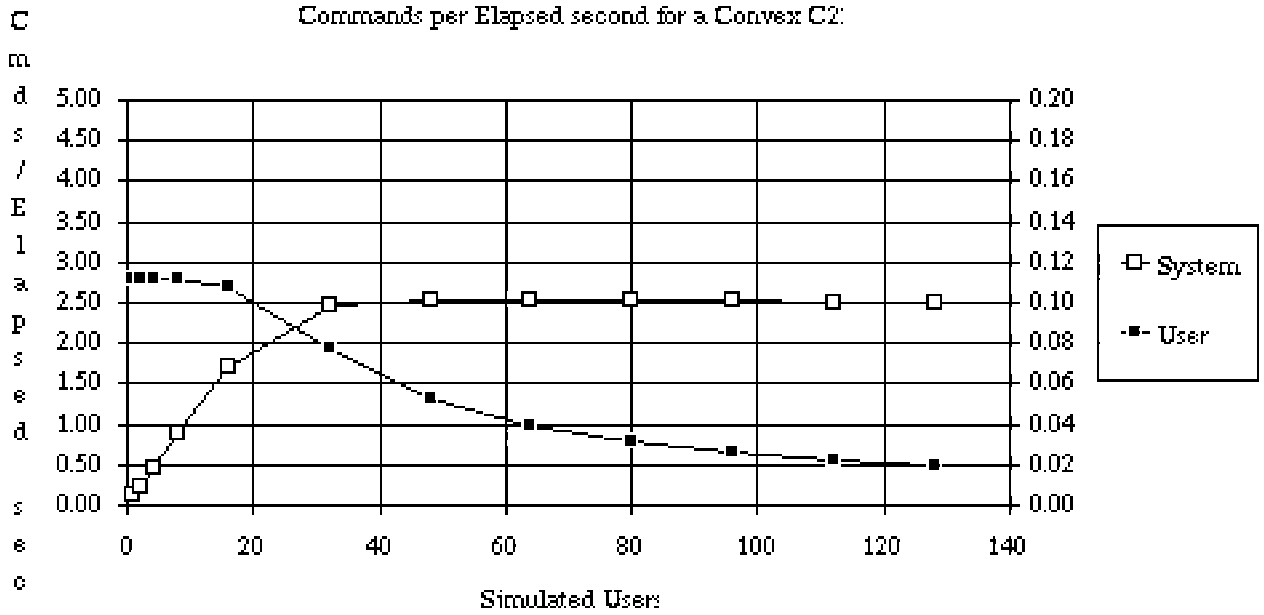
B.1 Convex C-220
A Convex C-220 was made available at the San Jose sales office. Standalone evening runs were made with Convex staff assistance. MUSBUS was initially transferred on tape, but electronic mail was possible for simple file fixes (although it took a better part of a day to arrive). Full Internet connectivity is now available.

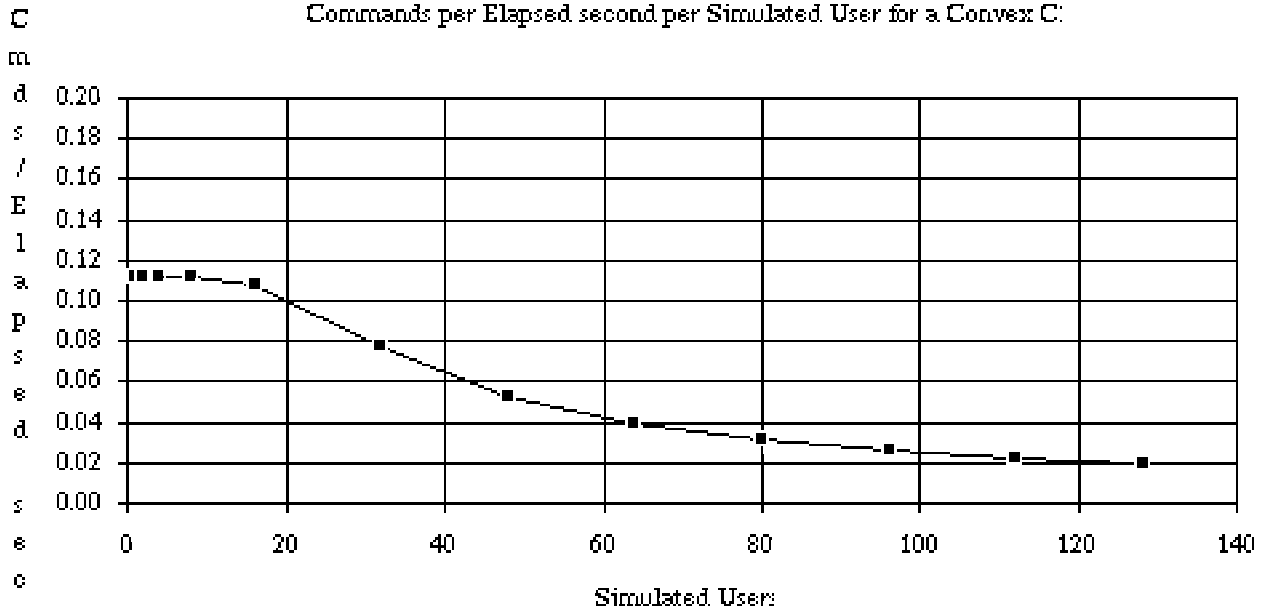
The first problem encountered was a `umask` variable which differed from protection in an open scientific environment. The Convex machine had industrial, commercial users on the machine during the day. This prevented full-scale daytime testing.

The operating system was modified for benchmark tests with increased kernel constants. Separate high-speed disks were added for our tests, and a symbolic link was used for a large /tmp area.

Convex staff helped the SPS team solve a workload problem which appeared on some systems. One of our workload's remote commands, an rsh [or rmsh on some systems (e.g. syntax inconsistency)], required the "-n" option on affected systems to direct any output to /dev/null.

Convex pages





B.2 Silicon Graphics 4D-380

The SGI 4D/380 is perhaps one of the easiest systems to test. The factory is physically close, and the machine is in a development environment like the NAS. We have access to other SGI machines as well, several 4D/340s for debugging, an Internet gateway which allows access back to Ames for files or communication.

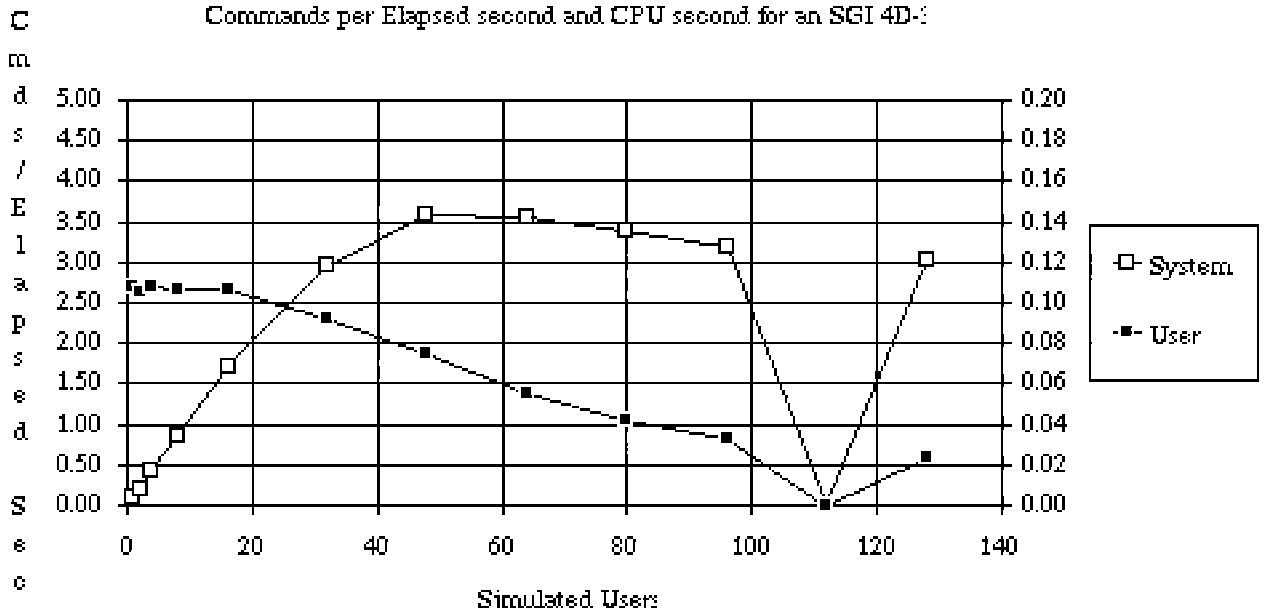
We can reboot the SGI machine any time. It has interesting performance tuning and measurement tools. SGI has several people to help tune systems, isolate the local area network, etc. Testers can make operating system tuning changes when necessary. SGI is very helpful. They have a large staff with MUSBUS experience, and we spent time talking to half a dozen people in three or four different groups using MUSBUS.

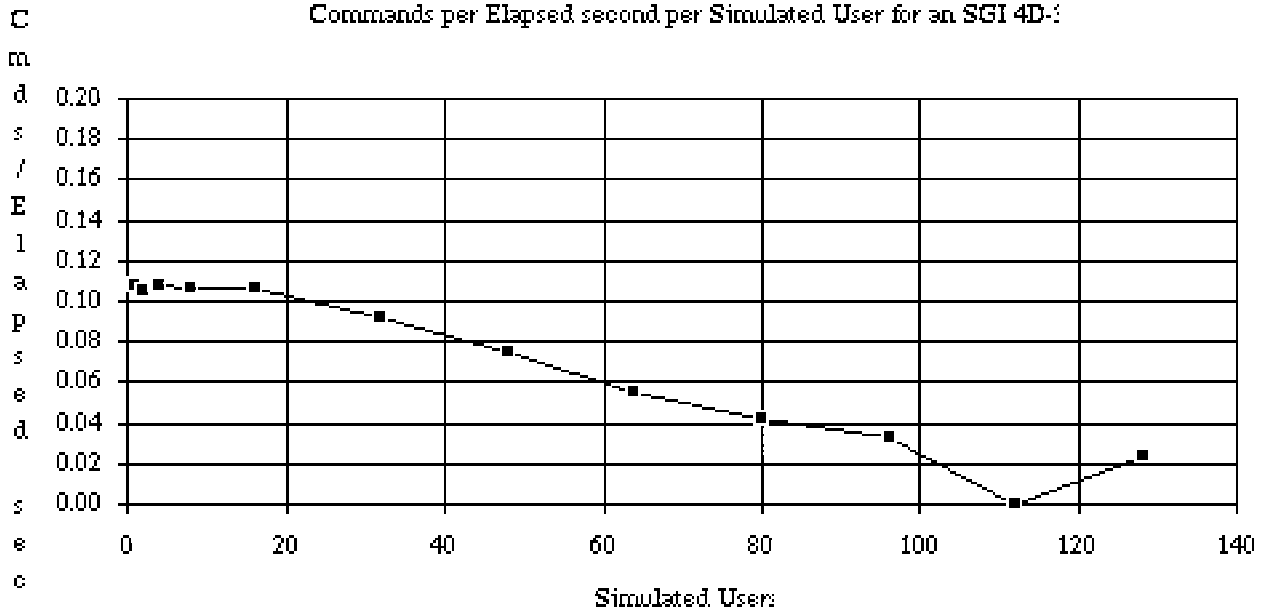
The SGI came close to completing the SPS workload, but frequently died during long tests leaving many asynchronous `sendmail` processes after terminating. The SGI also exhibited inconsistent problems formatting the output to the time.awk scripts. Beyond 64-users, timing data became "noisy." We did not have time to stop and analyze this problem but will inform SPEC and McDonell of the problem. (See the last paragraph in this section.)

Operating system configuration is a problem at SGI. Near the final test, we nearly received an absolutely "clean" machine off the assembly line with perfectly clean disks. While we were making our modifications, other groups were also benchmarking. The combined effect and loose configuration control which allowed any-time reboots, also make configuration control nearly impossible. This is a perfect example of "mixed blessing."

Notice a test anomaly in the data. After the test, a problem with SysVtime.awk was discovered which did not handle hour-long benchmarks. This problem was solved by changing the awk script. SPEC had independently also found this problem. The noisy data point was due to a floating-point overflow (division by zero) of improper timing.

SGI results


Commands per Elapsed second and CPU second for an SGI 4D-
Cmds/Elapsed Sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140
Simulated Users
System
User

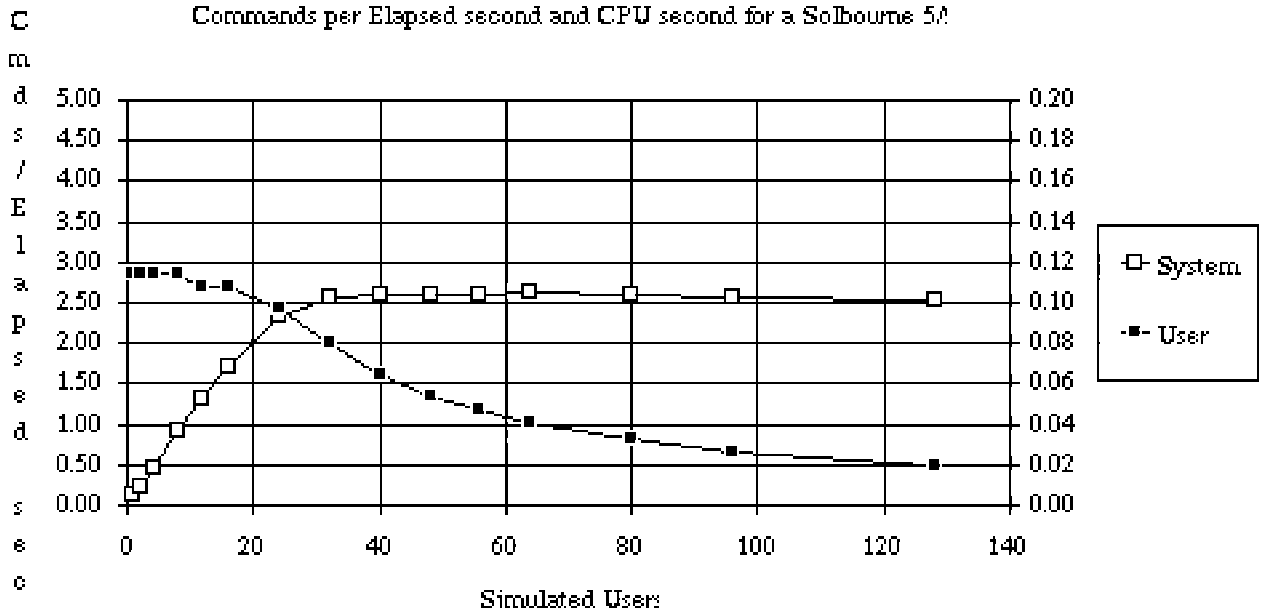

Commands per Elapsed second per Simulated User for an SGI 4D-
Cmds/Elapsed Sec
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140
Simulated Users

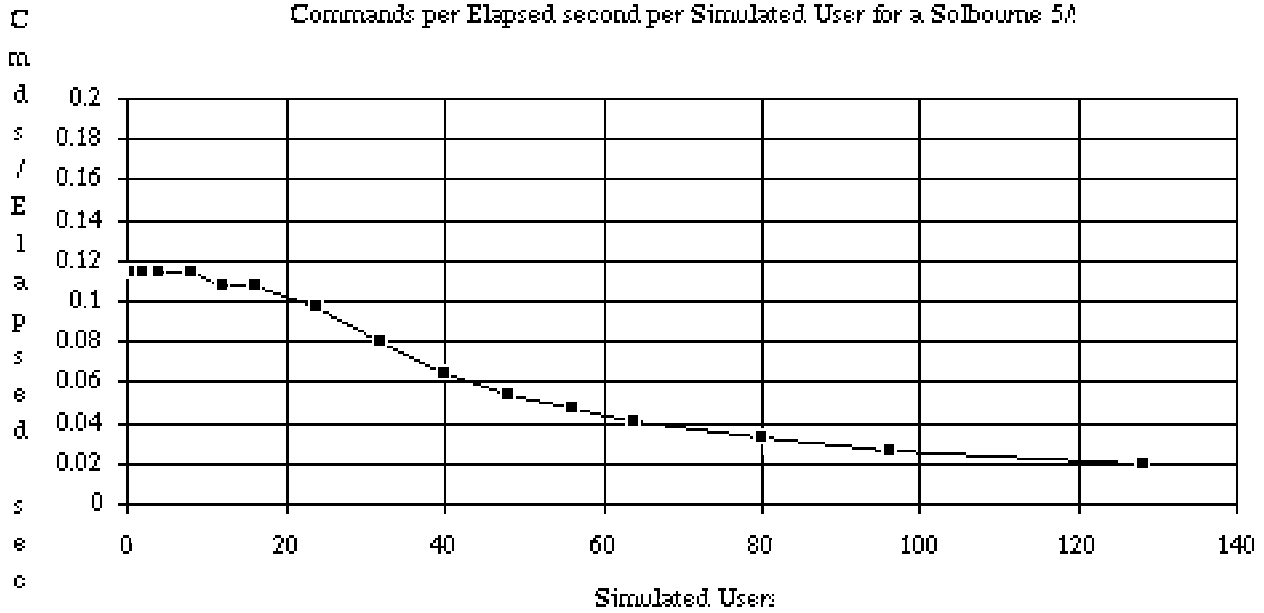
B.3 Solbourne

A two-processor Solbourne system was delivered to our machine room on the same day a SPEC meeting was held at MIPS. Solbourne was the first machine to complete test and measurement of our workload. This gave confidence that we could simulate 128 users. The machine died when ramping, because it would run out of diskspace [for mail], but individual tests ran fine. The owner of our particular CPU was coincidently attending the SPEC meeting and was quite knowledgeable about MUSBUS. He was an immense help.

Later, an 8-CPU machine in Colorado was made available to us via phone line. It was possible to reach their Internet gateway if we needed files or returning results. Reboots were easy considering the physical distance to the machine. We fell back to a front-end machine until the reboot was over. Noisy telephone lines sometimes made work difficult. The Internet was an excellent facility for Solbourne tests, but like Silicon Graphics and MIPS, they had a one-way gateway, i.e. communications out of Solbourne was acceptable, but communications into Solbourne was not.

Solbourne results

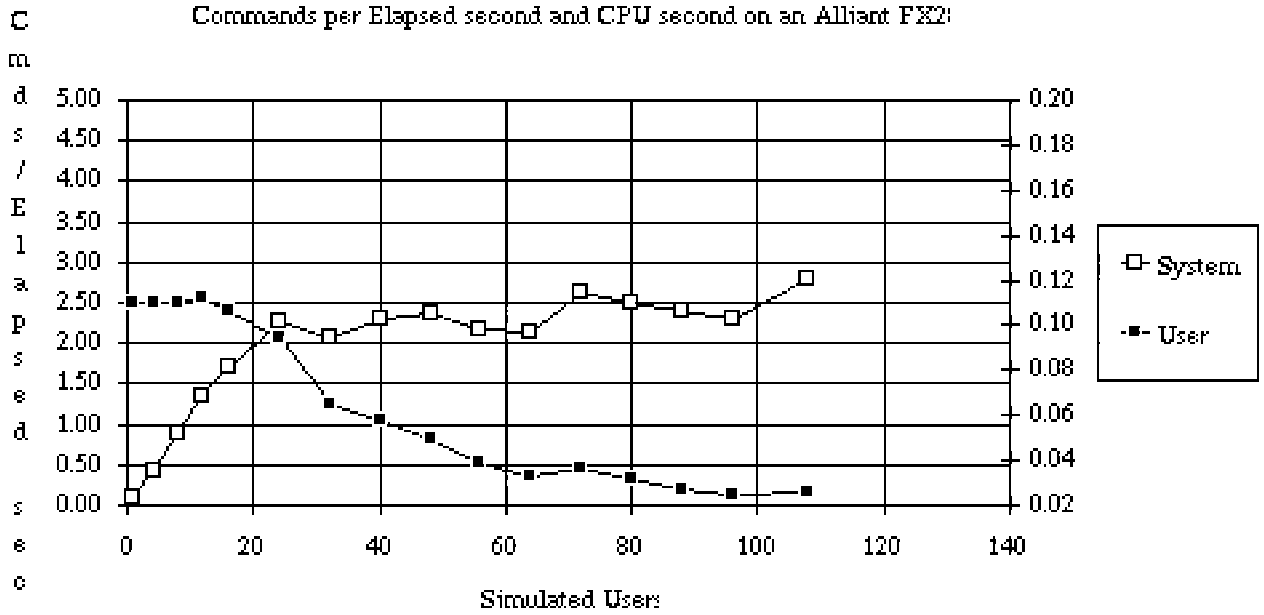


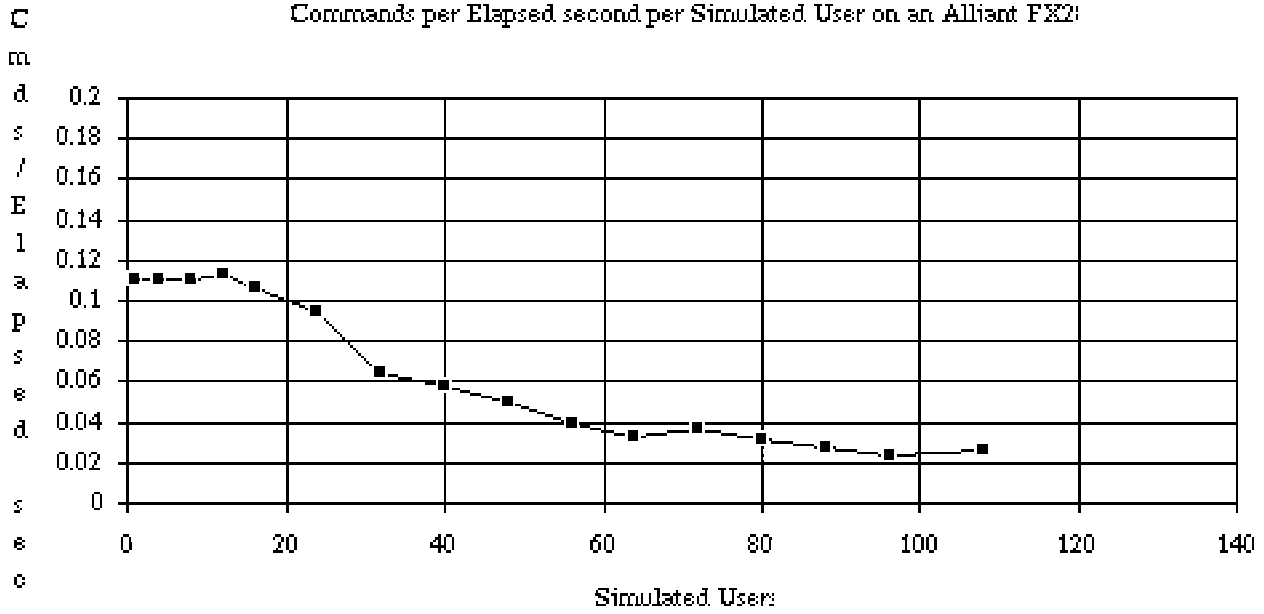


B.4 Alliant FX-2812

The Alliant computer was reached via telephone lines. A service office exists locally, but the sales force has an office in Los Angeles with the benchmarking machine located in Massachusetts. The Alliant staff made kernel changes and reboots, and later did some of the testing. Sending mail files around occasionally proved difficult. To date, some mail never reached Ames.

Alliant results





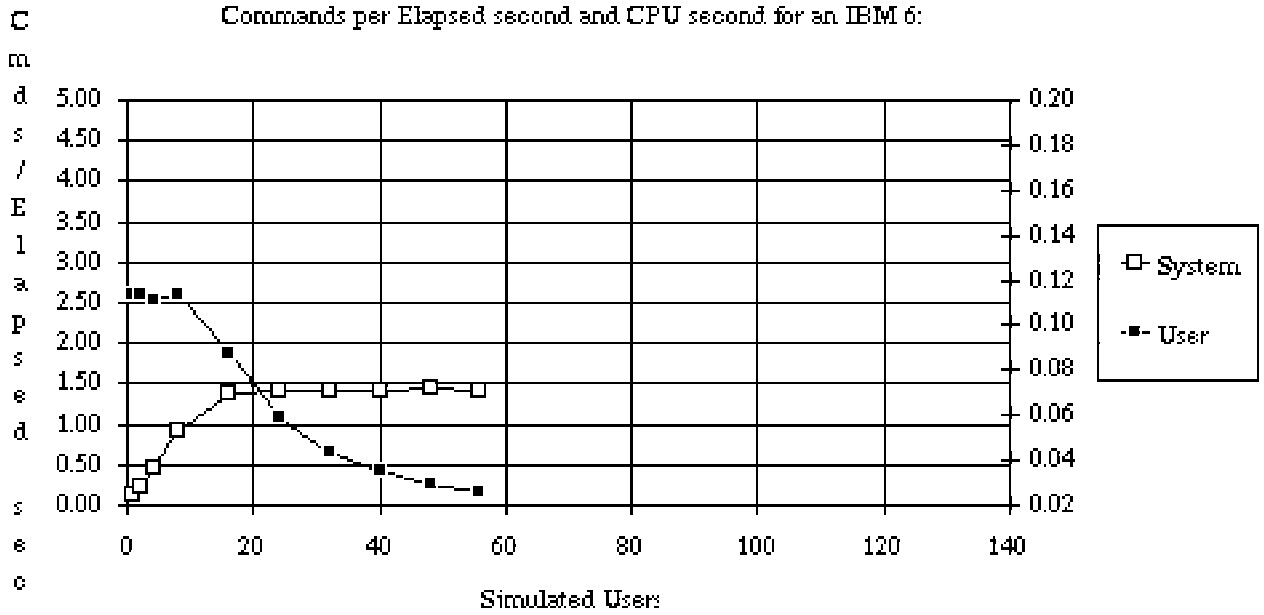
B.5 IBM RS/6000-320 and -530

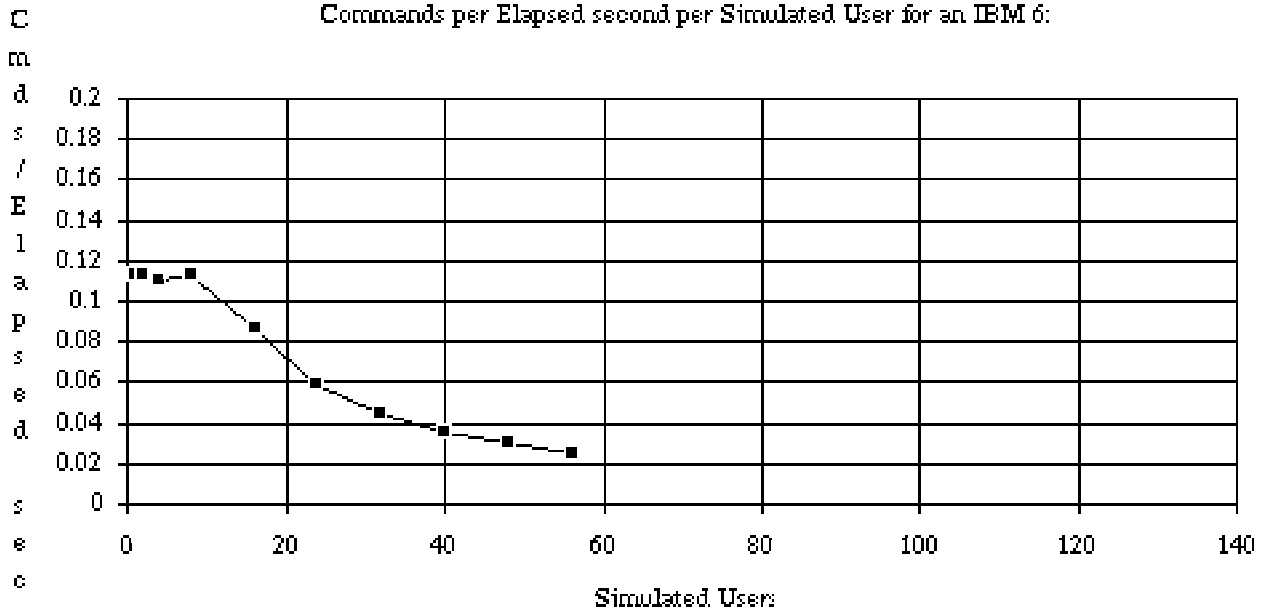
At first, IBM loaned us an RS/6000-320 for benchmarking. We did more operating system debugging than testing on this machine. Later, a model -530 was given to a NAS user and we were allowed access for local testing and development. The final test was on a model 530 at IBM's sales office in San Jose. All files were moved by cartridge tape. A byte-order problem required filtering all files through "dd conv=swab" (to or from NAS). Reboot was a simple button push.

The software improved from the earlier 320 versions of AIX. All AIX versions had IBM's new shadow security system. The disk hardware made screeching noises during the benchmarks and reboots. A special IBM program is used to configure and tune the system.

The local sales office involved other IBM Divisions: the RS/6000 development group in Austin, the IBM RS/6000 SPEC representative, and even the IBM 3090 supercomputer group. This latter group asked for copies of the NAS SPS workload for their use. In their words: "We want this workload, because it is perhaps unique in the world being one of the few UNIX based workloads on any mainframe or supercomputer."

IBM results


Commands per Elapsed second and CPU second for an IBM 6:
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
20
40
60
80
100
120
140
Simulated Users
System
User

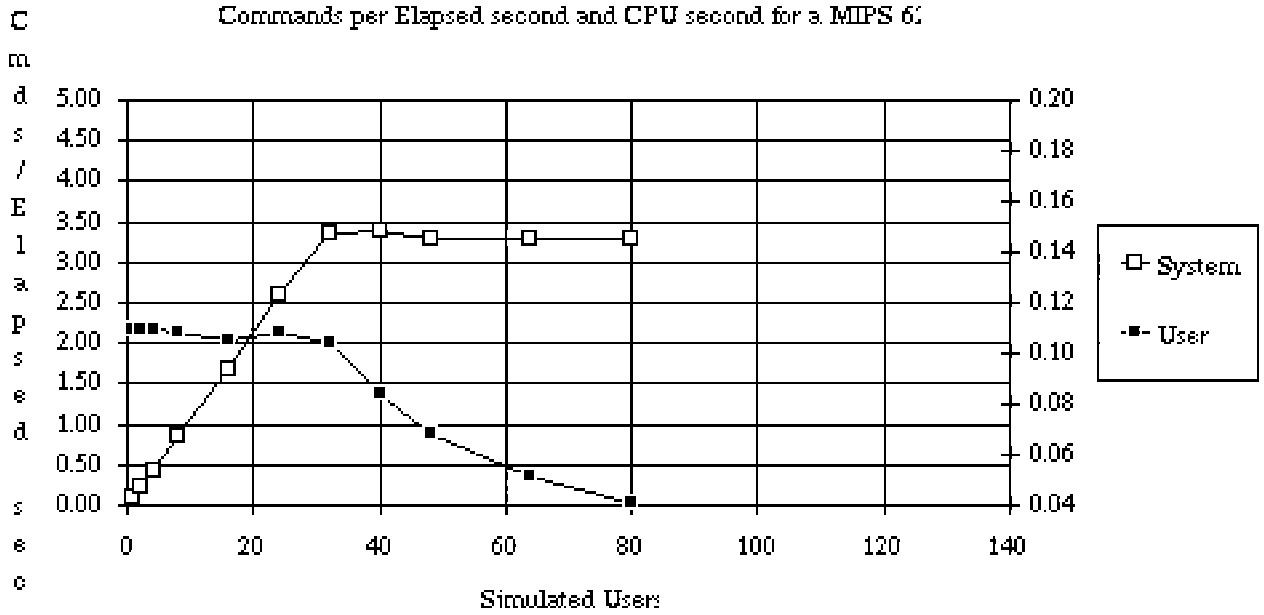

Commands per Elapsed second per Simulated User for an IBM 6:
Cmds/Elapsed sec
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100
120
140
Simulated Users

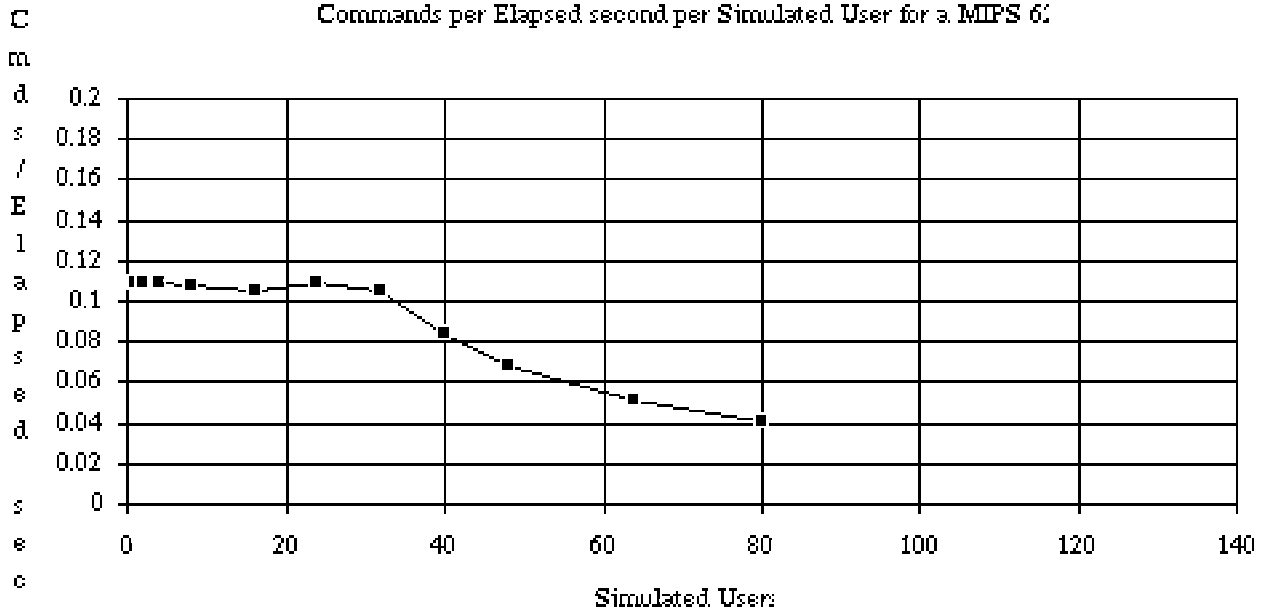
B.6 MIPS 6000

The MIPS 6000 is a uniprocessor, but unlike the other machines in this survey, it was the only ECL-based (rather than MOS) system. The system is a very fast single CPU. File transfer and remote login are possible through the MIPS gateway machine.

MIPS ported MUSBUS and our workload as well as system tuning. MIPS personnel also ran the tests since 6000 CPUs were a scarce commodity.

MIPS results

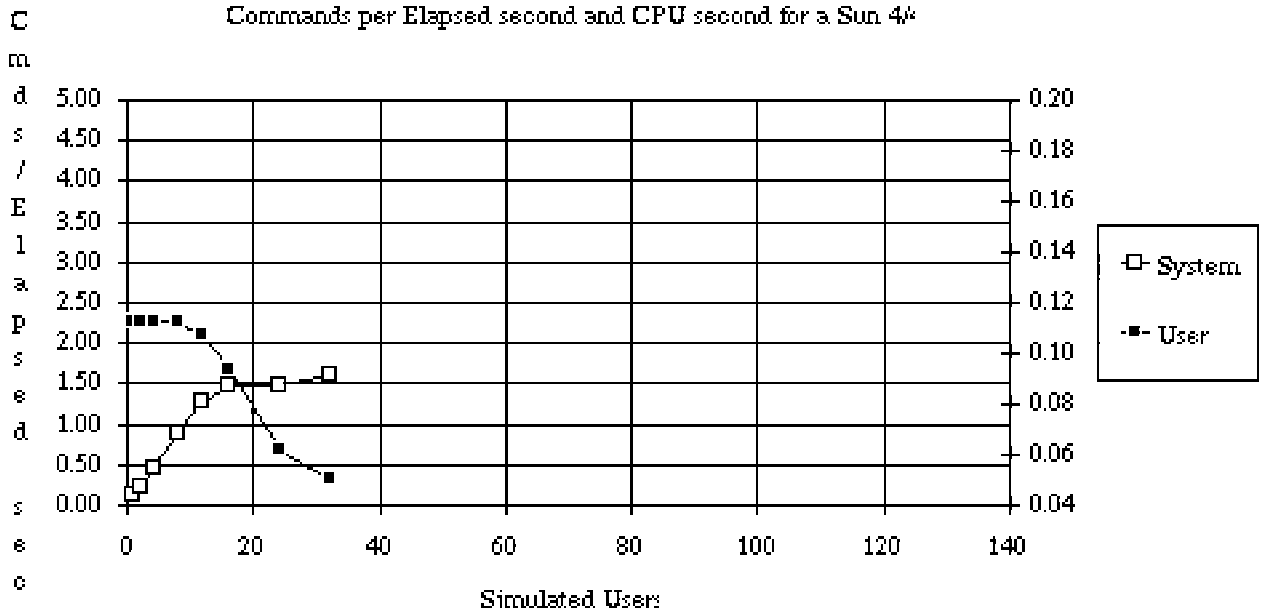


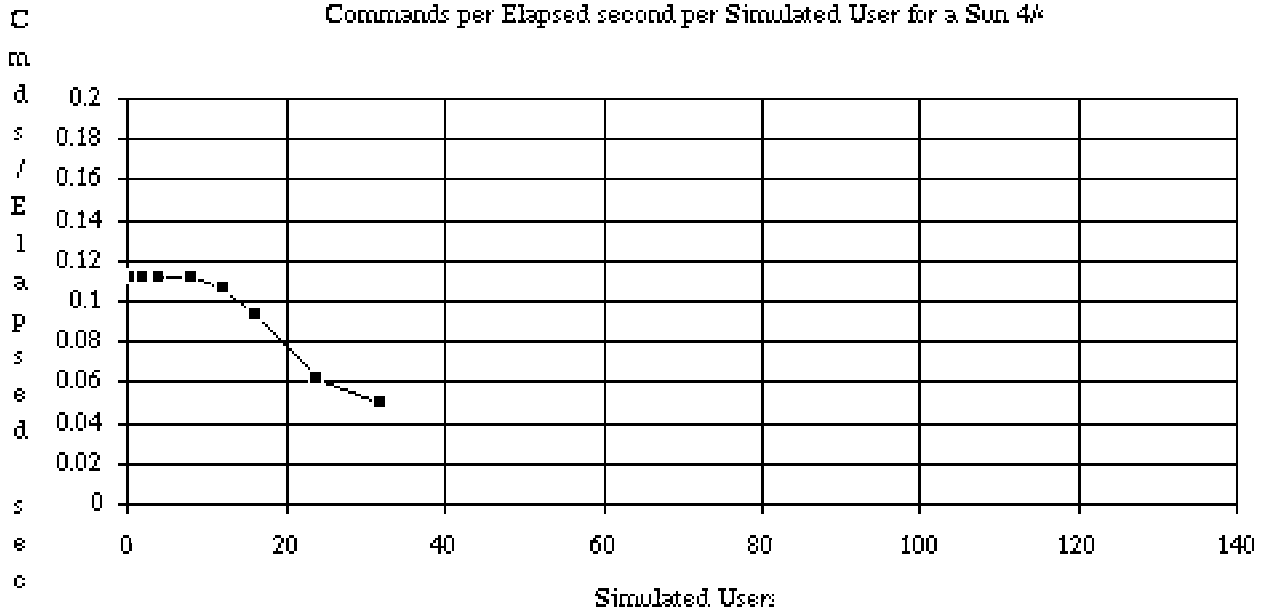


B.7 SUN 4/490

Several local NAS machines are available. SUN is planning an entry-level multi-microprocessor (SUN MP) sometime this year. An existing SUN CPU is included for comparison purposes.

sun490 results





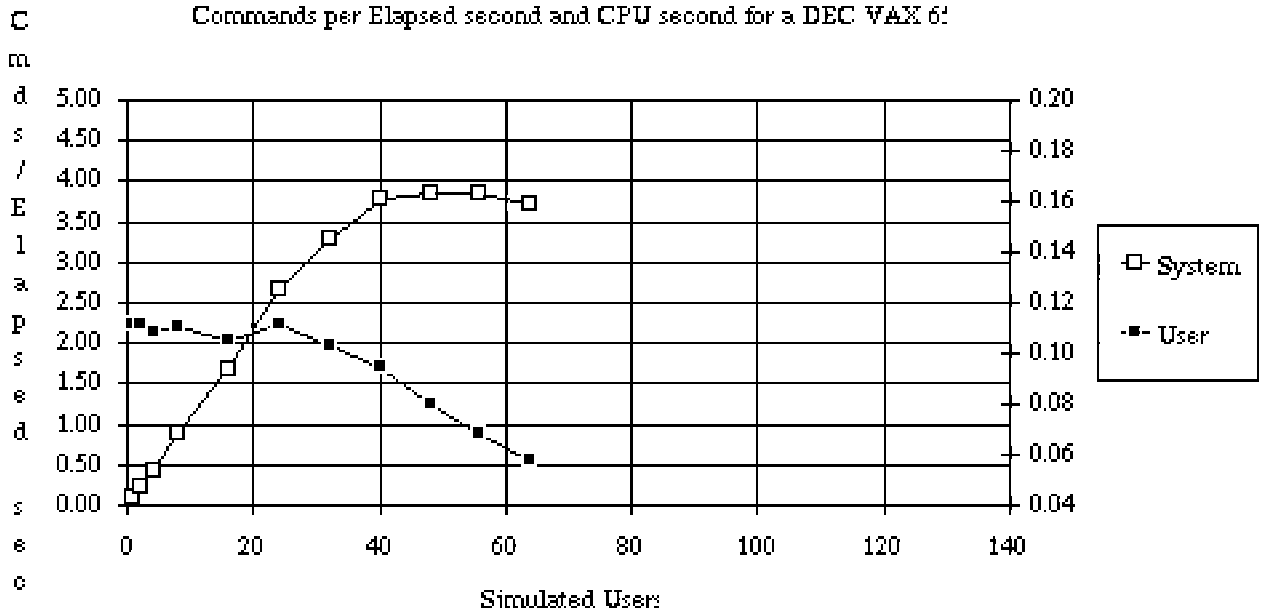
B.8 DEC 6000/9000

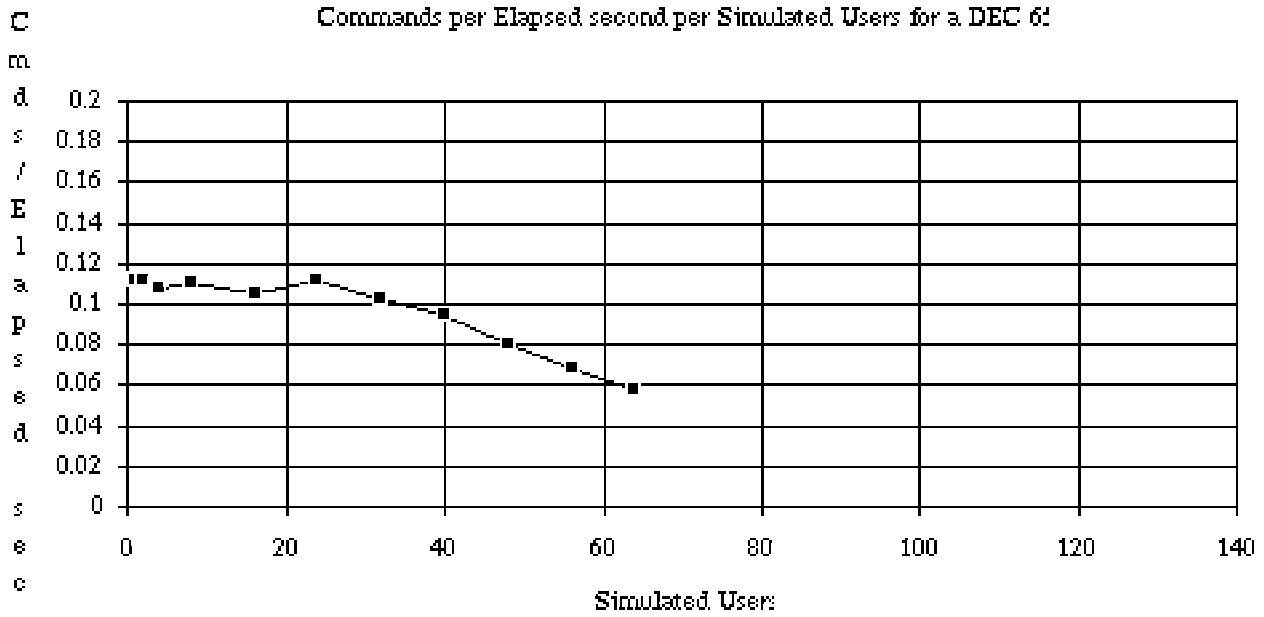
The first DEC VAX/9000 multiprocessor with vector units running Ultrix in alpha test was made available to us. DEC machines were available via dial-in, but the newness of this machine also had DEC people standing over our shoulders. DEC is intensively networked, so tape, mail or file transfers are possible.

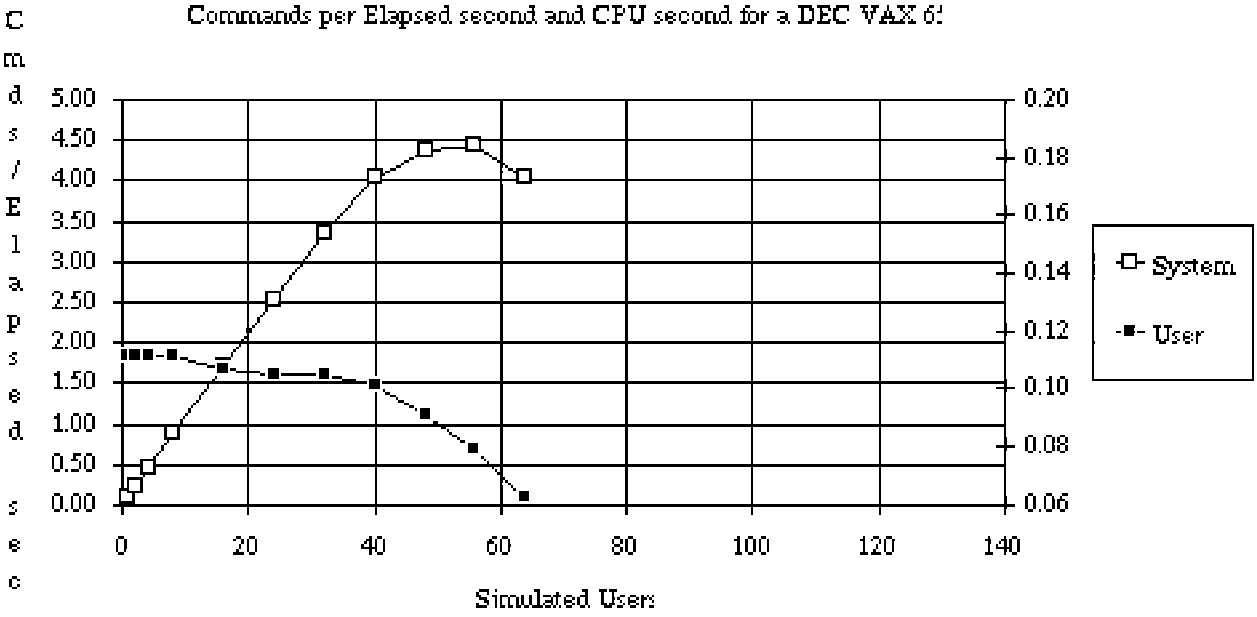
Additionally, VAX 6000s were also tested having a lower cost, vector, and multiprocessor options. The performance of the VAX 6000/560 was typical of the machine tested. The price-performance then becomes a distinguishing point for machines.

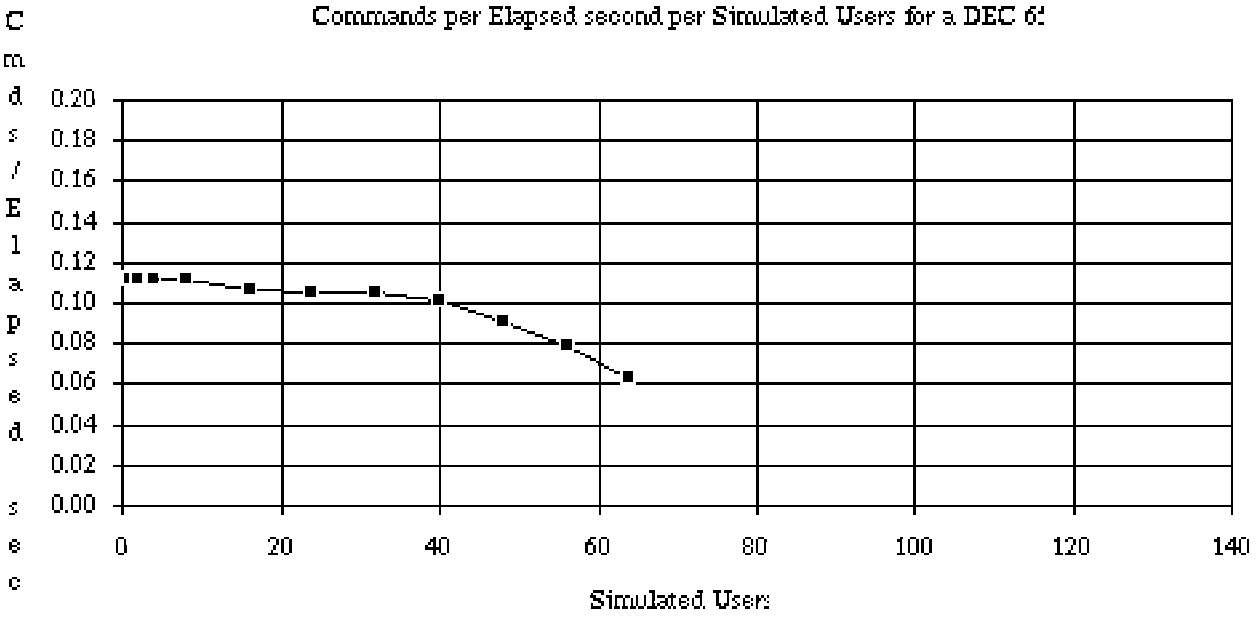
A VAX-11/780 was also tested for purposes of comparison. The performance of this graph is included at the end.

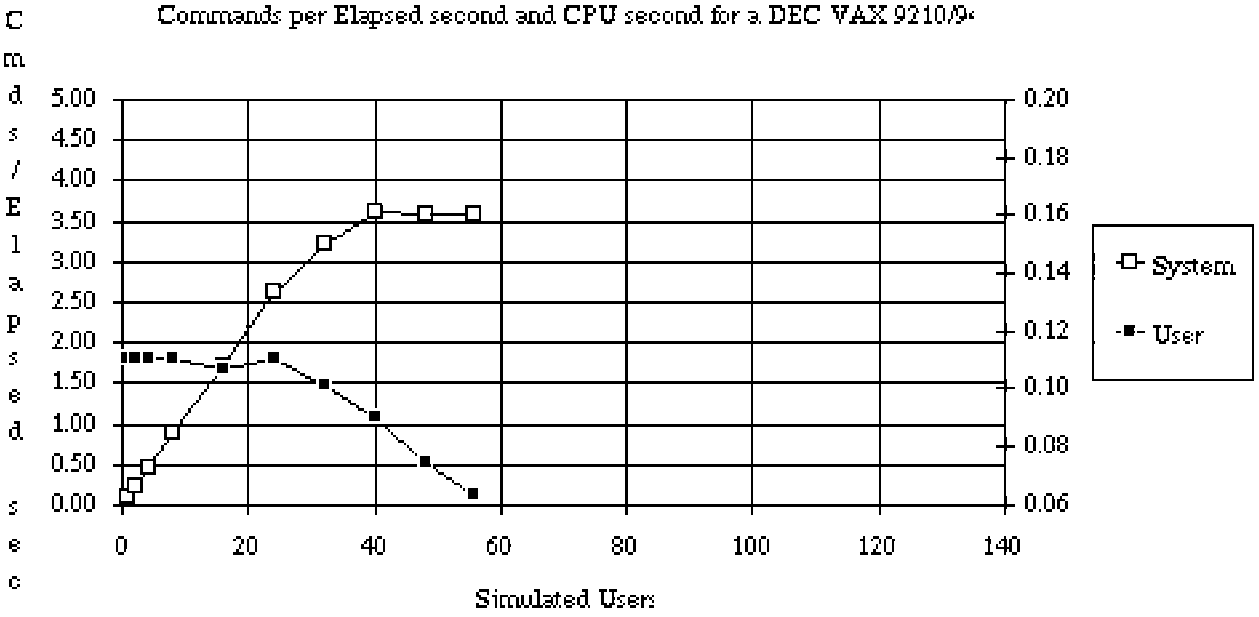
DEC Results

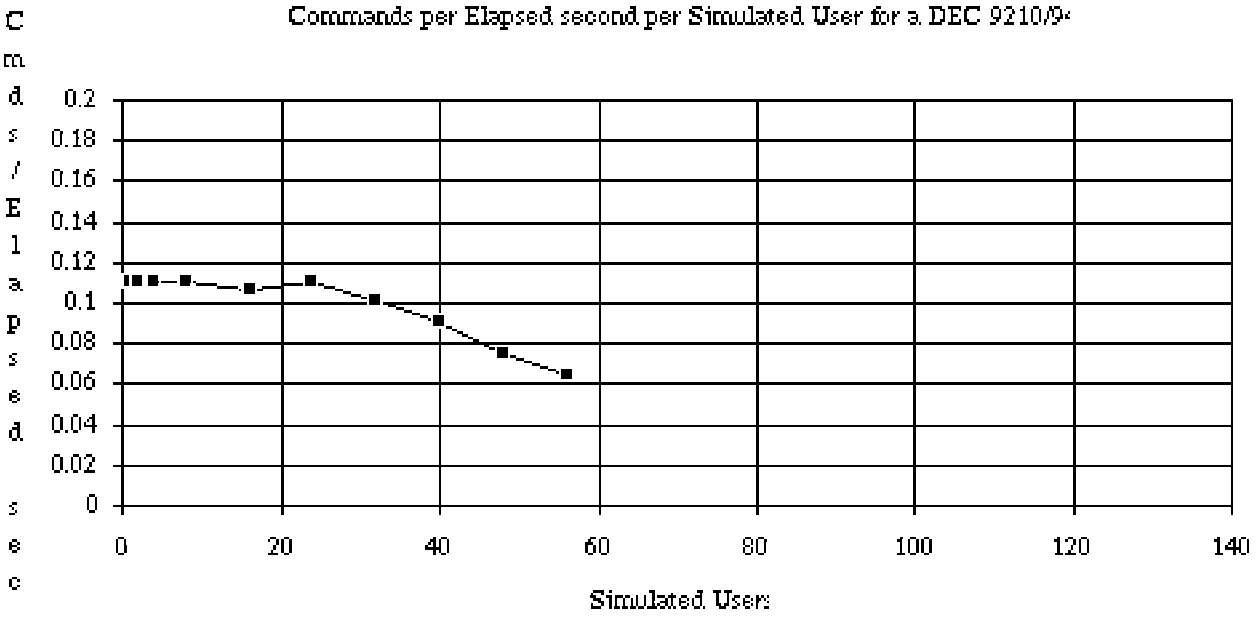

Commands per Elapsed second and CPU second for a DEC VAX 6!
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0
20
40
60
80
100
120
140
Simulated Users
System
User

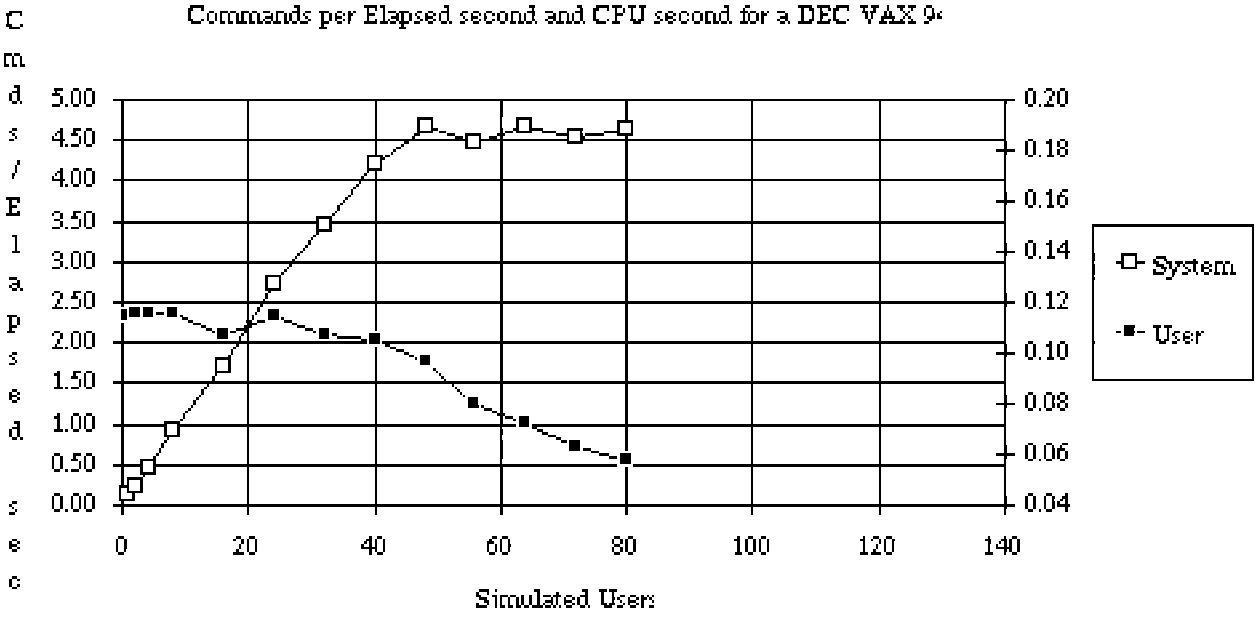

Commands per Elapsed second per Simulated Users for a DEC 6!
Cmds/Elapsed sec
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100
120
140
Simulated Users

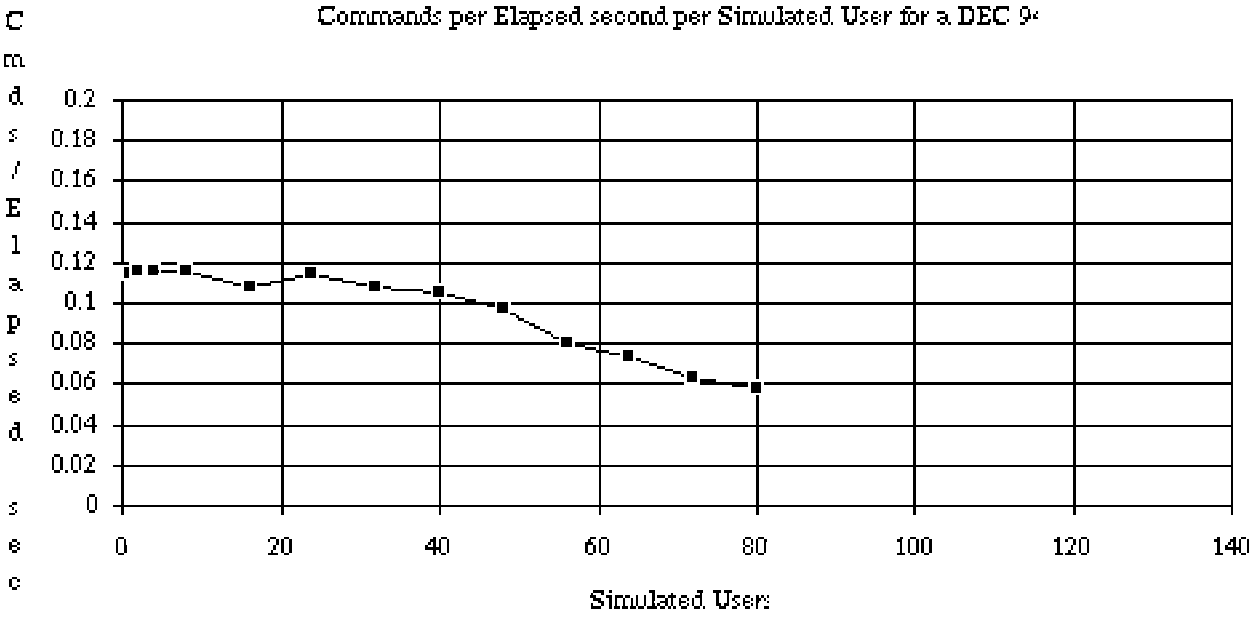

Commands per Elapsed second and CPU second for a DEC VAX 6!
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0
20
40
60
80
100
120
140
Simulated Users
System
User

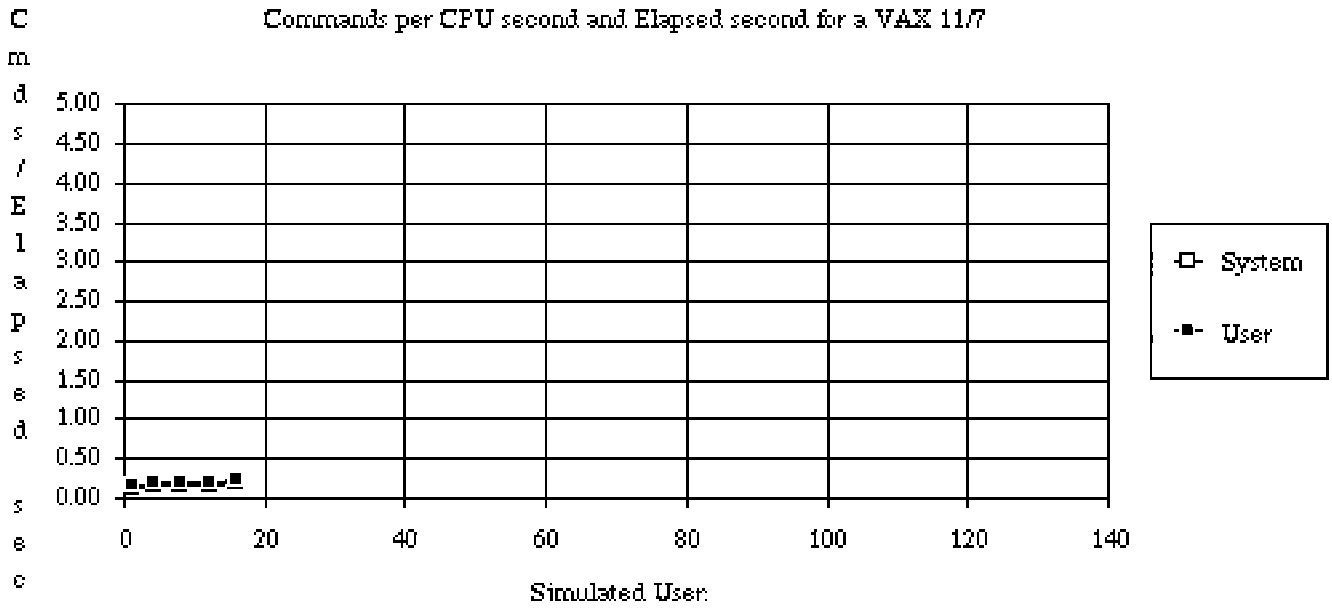

Commands per Elapsed second per Simulated Users for a DEC 6!
Cmds/Elapsed sec
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140
Simulated Users


Commands per Elapsed second and CPU second for a DEC VAX 9210/9
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0
20
40
60
80
100
120
140
Simulated Users
System
User


Commands per Elapsed second per Simulated User for a DEC 9210/9
Cmds/Elapsed sec
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100
120
140
Simulated Users


Commands per Elapsed second and CPU second for a DEC VAX 9
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0
20
40
60
80
100
120
140
Simulated Users
System
User


Commands per Elapsed second per Simulated User for a DEC 9
Cmds/Elapsed sec
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80
100
120
140
Simulated Users


Commands per CPU second and Elapsed second for a VAX 11/7
Cmds/Elapsed sec
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0
20
40
60
80
100
120
140
Simulated User
System
User

Sequent Symmetry
A copy of KENBUS and our workload were given to Sequent. We never heard from them again. Sequent has SPEC representation, and two Sequent systems are on base.

Partial ports to Cray Y-MP, BBN TC2000,
Partial MUSBUS ports without extensive work were made on available NAS machines: SGI Iris 4D/60, 70, and 320. The Cray-2 and the Cray Y-MP were tested as far enough to identify problems, but portability repairs were not made. A port was also tried on the BBN TC2000 located at the Massive Parallelism Computing Institute [MPCI] at the Lawrence Livermore National Lab [LLNL]. With some work, most ports would have run. Most porting problems are the result of BSD/System V OS blends.